Exploring how OpenMP can improve performance on applications

Dehui Yu
School of Computer Science
University of Ottawa
Ottawa, Canada K1S 5J6
dyu105@uottawa.ca

December 26, 2022

Abstract

OpenMP is the dominant programming model for shared-memory parallelism in C, C++ and Fortran due to its easy-to-use directive-based style, portability and broad support by compiler vendors. CPUs are now generally 4 or more cores, which is good news for computationally intensive programs. In the absence of GPU acceleration, consider using CPU multithreading for acceleration. This has great value in areas such as scientific computing. OpenMP is such a concurrent programming framework.My project is also investigating why OpenMP can become an industry standard API for shared-memory Programming. We've done a lot of literature review exploring why OpenMP stands out and why it's different from other parallel APIs. In addition, I used two practical examples, to explore how much more efficient parallelism using OpenMP can be compared to traditional serial computing. One example we use is based on the Monte Carlo method. Using C++ programming language, explore the effectiveness of using OpenMP parallel computing and compared with MPI. Our experimental results show that simply increasing the numbers of thread does not always lead to a continuously improvement in application performance. Also, our results indicate that MPI is slightly faster than OpenMP under the same application and number of threads.

1 Introduction

In this section, we introduce parallel computing in general and then introduce OpenMP - an industry standard API for Parallel programming , which is the API we use for performance analysis on parallel computing in our project. Then we outline the structure of the paper and our project goals.

1.1 Parallel Programming

Complex problems require complex solutions. Instead of waiting hours for a program to finish running, we can utilize parallel programming. Parallel programming helps developers break down the tasks that a program must complete into smaller segments of work that can be done in parallel. While parallel programming can be a more time intensive effort up front for developers to create efficient parallel algorithms and code, it overall saves time by leveraging parallel processing power by running the program across multiple compute

nodes and CPU cores at the same time.[11][10]

Parallel Computing refers to the process of using multiple computing resources to solve computing problems at the same time.[4] It is an effective means to improve the computing speed and processing power of computer systems. Its basic idea is to use multiple processors to solve the same problem, that is, the problem to be solved is decomposed into several parts, each part by an independent processor for parallel computation. A parallel computing system can be either a specially designed supercomputer with multiple processors or a cluster of several independent computers interconnected in some way. The data is processed by the parallel computing cluster and the result is returned to the user.

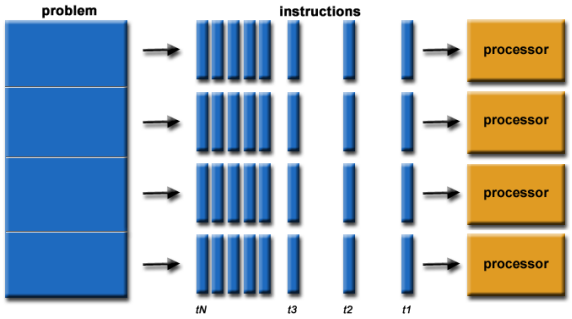

Figure 1: Parallel Computing

The purpose of parallel computing is to speed up computing. A descriptive definition of parallel computing is given in this paper: parallel computing refers to the process of decomposing an application into multiple subtasks on a parallel machine and assigning them to different processors, which cooperate with each other to execute the subtasks in parallel, thus accelerating the solution speed or solving the scale of the application problem.As Figure 1 shows, parallel computing, in general terms, is a computing architecture in which multiple processors simultaneously perform multiple, smaller computing tasks decomposed from a larger complex problem. In recent years, parallel computing has become a major paradigm in computer architecture, mainly in the form of multi-core processors.[1]

1.2 OpenMP - an API for parallel programming

In the previous sub-section, we mentioned that parallel computing has great potential for improving program performance. OpenMP is one of the interfaces to implement parallel computing and is widely used in the industry.

OpenMP was proposed by OpenMP Architecture Review Board and has been widely accepted as a set of Compiler Directive for the design of multi-processor programs in shared memory parallel systems. OpenMP supports programming languages including C, C++, and Fortran;The compilers that support OpenMp include Sun Compiler, GNU Compiler and Intel Compiler. OpenMp provides a high-level abstract description of parallel algorithms. Programmers specify their intentions by adding dedicated pragma to their source code, so that the compiler can automatically parallelize programs, including synchronous mutex and communication where necessary. When the pragma is ignored or the compiler does not support OpenMp, the program can be reduced to a normal (usually serial) program. The

code still works, but it cannot use multiple threads to speed up the program execution. OpenMP was created as a shared storage standard. It is an application programming interface (API) designed for writing parallel programs on a multiprocessor. It includes a set of compilation instructions and a library to support it. With dual-core and quad-core cpus now available, and six-core cpus already available, writing and running parallel programs on multiprocessors has become quite common.[3]

1.3 Project goals and achievements

Our project has four goals:

1. Conduct a detailed literature review to investigate why OpenMP is now an industry standard API for shared-memory programming and its real-world applications.
2. We implement the method of calculating PI using Monte Carlo proposed in paper[13] using parallel programming and compare the performance with serial programming.
3. We utilize block matrix for large-scale matrix operations traditionally used in parallel programming, and minimize the time complexity on large-scale matrix multiplication.
4. Investigate the connection and differences between OpenMP and MPI, and try to use MPI API in our application, then compare performance of MPI and OpenMP under the same application and number of threads.

After defined our project goals, we began our research. Some things we have successfully accomplished are:

1. We did a research survey on OpenMP. Study the industrial application of OpenMP and summarize it in Chapter 2. Readers can understand why OpenMP has become the industry standard and some interesting applications in the real world.
2. We successfully implement the method of calculating PI using Mote Carlo algorithms in OpenMP and MPI. And we compared the performance between serial programming and parallel programming. In addition, we found the application using OpenMP is slightly faster than using MPI under the same number of threads.
3. We implement block matrix multiplication using OpenMP and compare it with non-block parallel and sequentional implementation.

In addition, this is my first experience with parallel computing and C++. It was a very big accomplishment for me personally. I am grateful to this course and Prof. Frank. I have learned a lot from this course and I believe I will benefit from parallel programming in the future.

1.4 Paper outline

The structure of this article is as follows: In Chapter 1, we briefly discussed what parallel programming is and introduced OpenMP. In Chapter 2, we did our literature review and concludes why OpenMP is now the industry standard and introduces some practical application examples. Chapter 3, we discuss the problem statement. In Chapters 4 and 5, we built our own OpenMP application and analyzed its performance. In Chapter 6, we try to run our proposed program using MPI, and compared the performance with openMP. In the sixth chapter, we summarized our work and propose the future researches.

2 Literature Review

In this section, we conclude why OpenMP becomes the standard API for shared-memory programming and its real-world examples based on liteature review. The goal of this section is to present readers the importance of OpenMP and list relevent research related to OpenMP.

2.1 Why OpenMP becomes the standard API for shared-memory programming?

It makes more efficient, and lower-level parallel code possible.[2] By using OpenMP for parallel computing, programmers can focus on the logic of the program code rather than how to do parallel programming in the program. Here is a simple OpenMP application.

```
#include < stdio.h >

int main(void)
{
    #pragma omp parallel
    {
    printf("Hello, world.\n");
    }

  return 0;
}
```

Figure 2: "Hello World" application using openMP

When the master thread reaches this line: "pragma omp parallel", it forks additional threads to carry out the work enclosed in the block following the pragma construct. The block is executed by all threads in parallel. The original thread will be denoted as master thread with thread-id 0. This is a very simple parallel application, so the application cost for developers to learn is very low, which is one of the important reasons for OpenMP's popularity.

It can be used on various accelerators such as GPGP and FPGAs. General-purpose computing on graphics processing units(GPGPU) is the use of a graphics processing unit, which typically handles computation only for computer graphics, to perform computation in applications traditionally handled by the central processing unit.OpenMP has established itself as an important method andlanguage extension for programming shared-memory parallel com-puters. There are several advantages of OpenMP as a programming-paradigm for GPGPUs.[9]

1. Incremental parallelization of applications, which is one of OpenMP's features, can add the same benefit to GPGPU pro-gramming.

2. In order to make better use of the highly parallel computing unit of GPU, loop-level parallelism in OpenMP can be well expressed, thus speeding up the application of data parallel computing.

3. The relationship between the master thread operating in the host CPU and a pool of threads in a GPU device is nicely represented by the idea of a master thread and a pool of worker threads in OpenMP's fork-join paradigm.

It has scalability comparable to MPI on shared-memory systems, but no need to deal with message passing as MPI does. OpenMP is an API for parallel programming in shared memory. Unlike MPI before it, OpenMP is thread-level parallel, which is a bit lighter than MPI's process-level parallel. More importantly, MPI parallelism requires a complete rewrite of the entire program, whereas converting a serial program into OpenMP parallelism may require only a few changes. And gcc supports OpenMP natively, so there is no need to install a runtime environment and runtime library like MPI.[14]

2.2 Real-world examples using OpenMP

2.2.1 Parallel computation of a dam-break flow model using OpenMP on a multi-core computer

Dam construction projects serve people greatly, but they can also be dangerous. Even though there is extremely little chance that a dam will fail, if it does, there might be significant loss of life and property . As a result, research on dam-break flows is required, as well as the development of a mathematical model to predict their development, in order to give decision-making data for the planning of flood management.

One sort of rapidly fluctuating flow is the evolution of dam-break flows, which frequently takes place in dry riverbeds. Potential models must resolve computations involving discontinuous flows and wet/dry borders. An adequate solution approach and a fine mesh are often needed to increase simulation accuracy. Large data sets, however, significantly lengthen the model runtime, which limits the model's effectiveness. An average CPU would require around 9 hours to model the flow of a flood wave over a 62 km2 region utilising 708,864 grid cells. Therefore, the fundamental issues of simulation speed and precision must be properly resolved in any modelling technique, particularly if the customer requires real-time forecasts to be used for hazard planning.

This paper[16] introduced a parallel algorithm based on OpenMP to reduce the computation time of the flood time model.Parallel computation may be accomplished by adding OpenMP instructions to the circulation calculations in a time step, as shown in Figure 3. Grid cell flow and source calculations are executed by each thread, which is spread with distinct grid cell computations. The velocity and water depth may be updated since the computation result is immediately placed into the memory structure. The only variation in the flowchart of the parallel model solution from the serial one is the addition of certain OpenMP instructions to the time step loop. As a result, using OpenMP for parallel computation in this type of paradigm is simple.

The results demonstrate that the dam breach flow simulation accuracy and speed are improved by parallel calculation, and that a 16-core computer can simulate a 320-hour flood process in 1.6 hours. This results in an acceleration factor of 8.64. The model with more computing involved exhibits superior efficiency and higher acceleration, according to further investigation. In addition, the model scales well, and as the number of processing cores

rises, the acceleration ratio does as well. On a single computer, the parallel model based on OpenMP may simulate the flow of a dam breaking in a broad watershed by fully utilising the multi-core CPU.

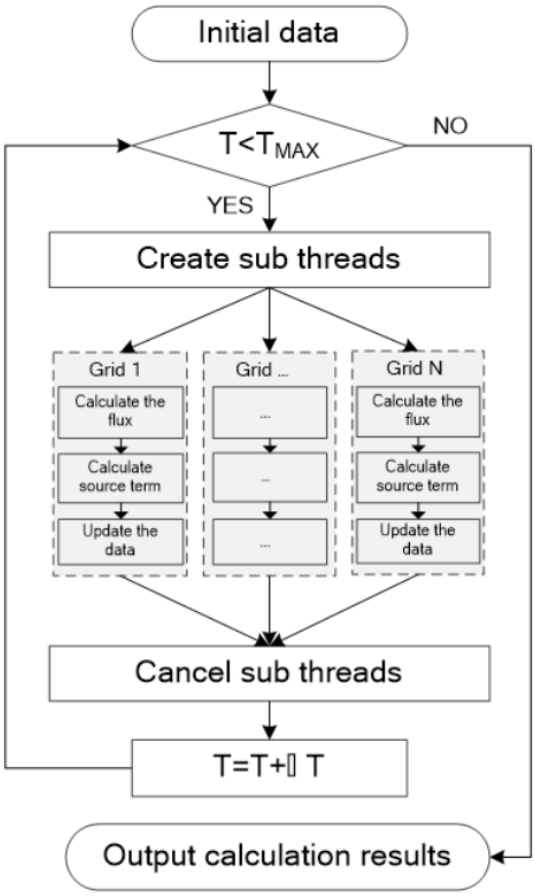

Figure 3: Flowchart of parallel computing in the dam-break model.[16]

2.2.2 Multicore Image Processing with OpenMP

This article[12] describes the use of OpenMP to make multithread image processing applications and take advantage of multicore general-purpose CPUs. They apply openmp to the Image Warping, as shown in Figure 4. An image warp is a spatial transformation of an image and is commonly found in photo-editing software as well as registration algorithms. In this example, they apply a "twist" transformation of the form.

$$
\begin{aligned}
x' &= (x - c_x)\cos\theta + (y - c_y)\sin\theta + c_x \\
y' &= -(y - c_y)\sin\theta \\
&\quad + (y - c_g)\cos\theta + c_y,
\end{aligned}
$$

The common variables in the code include the width and height of the image, as well as the original and altered images.where [cx, cy]T is the rotation center, =r/2 is a rotation

angle that increases with radius r,[x,y]T ‘is the point before transformation, and [x,y]T is the point after transformation. The effect of this image transformation in shown in Figure 4.

Figure 4: Flowchart of parallel computing in the dam-break model.

They used OpenMP to implement this transition; Figure 5 contains the code. The original image, the altered image, and the image's width and height are all shared variables in the code. The function iterates over the converted image's pixels [x,y]T, transferring them to the original image using the inverse transform of (1). The pixels in the original image are bilinearly interpolated. Each thread needs its own x, y, index, radius, theta, xp, and yp variables. We utilise the shared clause since the parallelized code initialises these variables. The outer loop (over yp) will be multithreaded by OpenMP using static scheduling.

```
int x, y;
#pragma omp parallel for \
shared(inputImage, outputImage, structuringElement, width, height) \
private(x, y) schedule(dynamic)
for (y = 0; y < height; y++) {
  for (x = 0; x < width; x++) {
       int index = x + y*width;
       if (inputImage[index]) {
          if (FullFit(inputImage, x, y, structuringElement))
             outputImage[index]=1;
          else
             outputImage[index]=0;
       }
  }
}
```

Figure 5: Parallelized code for the image warp.

On a 512 × 512 image, and using a quad-core 2.4 GHz CPU, the single-threaded code requires 62.2 ms to process the image, while the multithreaded code requires 17.7 ms, corresponding to a 3.5× speedup. The speed comparison between single-threaded and multithreaded can be seen intuitively in Figure 6.

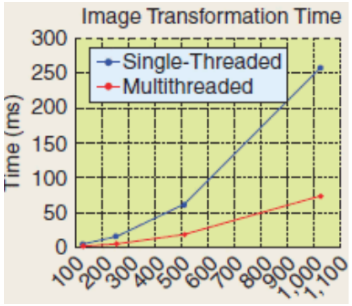

Figure 6: Image Transformation Time

3 Problem Statement

In chapter 2, literature review, we discussed why OpenMP is so popular and its practical application in the scientific research field. And we can see OpenMP has great potential on performance improvement. It's necessary to invegistate Since this is the first time for us to use OpenMP and parallel programming, we also want to prove the effectiveness of OpenMP through our own experiments. We took two mathematical applications and tried to convert them to parallel algorithms using OpenMP. Aiming at our project, we propose the following research questions and conduct two case studies according to them.

1. How to calculate PI value by using Monte Carlo algorithm? And how can we integrates OpenMP with Monte Carlo?
2. How does OpenMP parallel computing improve program running efficiency compared with serial program?
3. How to use OpenMP for matrix multiplication? And how to use block matrix operation to improve OpenMP program efficiency?
4. Does more threads always improve the efficiency of application?
5. It the answer for question 4 is no. What is the optimal number of CPUs for an application?
6. What is the difference between OpenMP and MPI? Will MPI API perform better than OpenMP API?

In chapter 2, we found there are some practical applications using OpenMP for parallel programming, and it seems that by adding more threads can always lead to performance improvement. We aim to answer the proposed questions by our own experiment, and we discussed our results in chapter 4 and 5.

4 Case Study 1: Monto Carlo - Calculating PI

4.1 Algorithm introduction

Monte Carlo method is also known as statistical simulation method, statistical test method. It is a numerical simulation method which takes probabilistic phenomena as the research object. It is a method of calculating unknown characteristic quantity by obtaining statistical

value according to sampling survey.[7] The law was named to indicate the random sampling nature of Monte Carlo, Monaco's famous gambling destination. Therefore, it is suitable for computational simulation test of discrete system. In the simulation, the stochastic characteristics of the system can be simulated by constructing a probabilistic model similar to the performance of the system and conducting random experiments on the digital computer.[8]

The basic idea of Monte Carlo method is: in order to solve a problem, first establish a probabilistic model or random process, make its parameters or numerical characteristics equal to the solution of the problem: then through the observation of the model or process or sampling test to calculate these parameters or numerical characteristics, and finally give the approximate value of the solution. The accuracy of the solution is expressed as the standard error of the estimate. The main theoretical basis of Monte Carlo method is the theory of probability statistics, and the main means are random sampling and statistical experiment.[6]The basic steps for solving practical problems with Monte Carlo method are as follows:

1. According to the characteristics of the actual problem. Construct a simple and easy to implement probability statistical model. So that the solution is exactly the probability distribution or mathematical expectation of the problem;

2. The sampling methods of random variables with different distributions in the model are given.

3. After statistical processing of the simulation results, the statistical estimate and precision estimate of the solution are given.

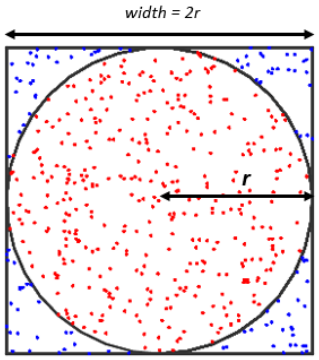

Figure 7: Estimating Pi using the Monte Carlo Method

One method to estimate the value of $(3.141592\ldots)$ is by using a Monte Carlo method.[13] This methods consists of drawing on a canvas a square with an inner circle. We then generate a large number of random points within the square and count how many fall in the enclosed circle, as shown in Figure 7.

We know that area of the square is $4r^2$ unit sq while that of circle is πr^2 The ratio of these two areas is as follows :

$$\frac{\text{area of the circle}}{\text{area of the square}} = \frac{\pi r^2}{4r^2} = \frac{\pi}{4}$$

In other words, if there are a large number of generated points, the ratio can be presented as follows: It can be seen that it is completely feasible to calculate PI by simulating points

$$\frac{\pi}{4} = \frac{\text{no. of points generated inside the circle}}{\text{total no. of points generated or no. of points generated inside the square}}$$

based on Monte Carlo, but it requires a large number of points to make it accurate.

4.2 Experimental environment

Our experimental platform consists of one cloud machines running Ubuntu 18.2 that has 16 vCPU, the max thread is 16. It's virtual machine from AWS EC2 platform. The Max thread is 16. It's hosted on AWS EC2 platform. The price is 0.796 per hour.

4.3 Implementation with OpenMP

The main function code of our program is as follows:

```
int main()
{
    std::cout.precision(20);
    std::cout << "Hello_OpenMP_PI_Program" << '\n';
    int numProcs = omp_get_max_threads();
    std::cout << "max_threads" << numProcs<<'\n';
    std::cout << calc_pi1(MAX_N) << '\n';
    std::cout << calc_pi2(MAX_N) << '\n';
    std::cout << calc_pi4(MAX_N) << '\n';
    std::cout << calc_pi8(MAX_N) << '\n';
    std::cout << calc_pi16(MAX_N) << '\n';
    return 0;
}
```

Figure 8: Main function-using Monto Carlo calculating PI

In our program, we tested the results of dropping 10 million, 20million and 30million points respectively, which is enough to calculate the PI value accurately. calc_pi1 means that the calculation is single-CPU, calc_pi2 means that the calculation is dual-CPUs, and so on. We tested a total of 5 scenarios: single CPU (no OpenMP), dual CPUs, 4 CPUs, 8 CPUs, and 16 CPUs, to compare the performance impact of different number of CPUs on our application.

Figure 9 shows how we calculate PI values by simulating scattered points. At the same time, we use omp built-in functions(omp_get_wtime()) to calculate the computing time under this environment.

4.4 Experimental Evaluation

We summarized the time duration into a linear graph, as shown in Figure 10. From the figure, we can find three things:

```
23  double calc_pi1(const long N)
24  {
25      float startTime1 = omp_get_wtime();
26      const double N1 = 1.0 / N;
27      double res = 0.0;
28
29  #pragma omp parallel num_threads(1)
30  {
31      double tmp = 0.0;
32
33  #pragma omp for
34  for (long i = 0; i < N; i++)
35      {
36          const double x = (i + 0.5) * N1;
37          tmp += 4.0 / (x * x + 1);
38      }
39          res += tmp;
40      }
41      float endTime1 = omp_get_wtime();
42      printf("For one thread time: %f\n", endTime1 - startTime1);
43      return res / N;
44  }
```

Figure 9: Main function-using Monto Carlo calculating PI

1. When N improves, the convergence becomes slower.
2. By simply adding more threads will not always improve application performance.
3. The best thread for this application is two thread or four thread.

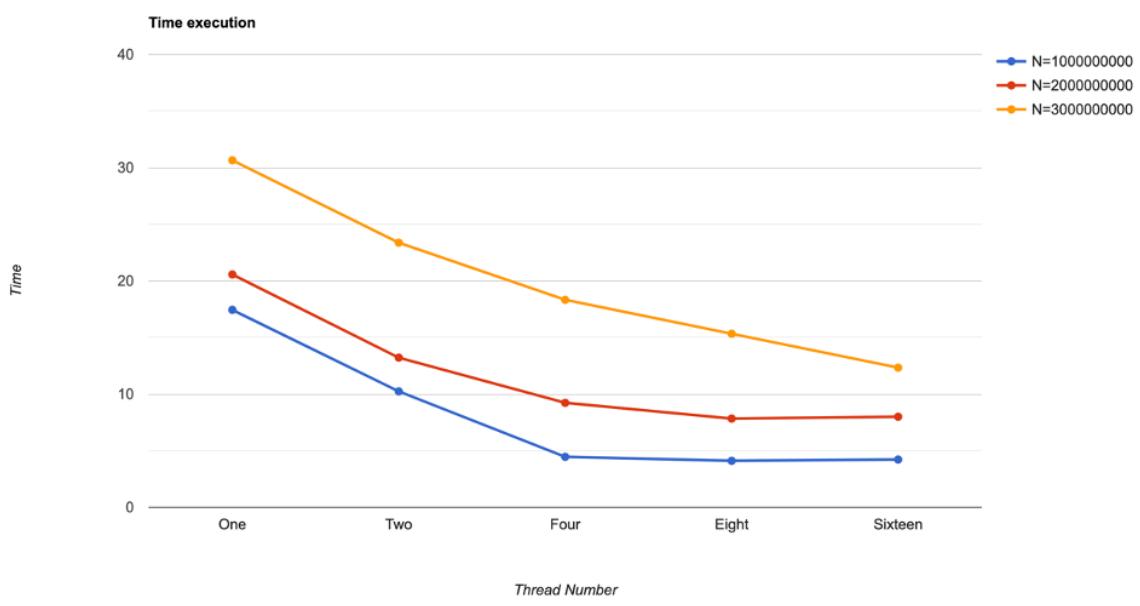

Figure 10: Time duration under different cores

From above, the second and third things answer questions 4 and 5 in Chapter 3, respectively.

5 Case Study 2: Matrix Multiplication using OpenMP

5.1 Introduction

In mathematics, particularly in linear algebra, matrix multiplication is a binary operation that produces a matrix from two matrices. For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix, known as the matrix product, has the number of rows of the first and the number of columns of the second matrix. The product of matrices A and B is denoted as AB. Below there are two matrices.

$$\mathbf{A} = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{pmatrix}$$

If we want to calculate the multiplication of A by B, the result can be expressed as the following:

$$\mathbf{C} = \begin{pmatrix} a_{11}b_{11} + \cdots + a_{1n}b_{n1} & a_{11}b_{12} + \cdots + a_{1n}b_{n2} & \cdots & a_{11}b_{1p} + \cdots + a_{1n}b_{np} \\ a_{21}b_{11} + \cdots + a_{2n}b_{n1} & a_{21}b_{12} + \cdots + a_{2n}b_{n2} & \cdots & a_{21}b_{1p} + \cdots + a_{2n}b_{np} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}b_{11} + \cdots + a_{mn}b_{n1} & a_{m1}b_{12} + \cdots + a_{mn}b_{n2} & \cdots & a_{m1}b_{1p} + \cdots + a_{mn}b_{np} \end{pmatrix}$$

In our application, the original matrix is automatically generated. Through the calculation of complex matrix, to explore the efficiency of OpenMP for improvement.

5.2 Experimental environment

The experimental environment is the same as in Case Study 1, our experimental platform consists of one cloud machines running Ubuntu 18.2 that has 16 vCPU, the max thread is 16. It's virtual machine from AWS EC2 platform. The Max thread is 16. But in this case we have been using the maximum CPUs (16 cores) to explore the efficiency, and compared with single CPU.

5.3 Implementation with OpenMP

First, we use random functions to generate matrices of different sizes. We generated matrices with different sizes: 128*128, 256*256, 512*512, 1024*1024, and 2048*2048. The generated matrix code is shown in Figure 11. We then run the for loop twice to perform the operation between the elements of the two matrices. The complete source code is available on my GitHub.

5.4 Experimental Evaluation

In our second example, we always use the maximum number of cores (16 cores) for the calculation, and the running time of matrices of different sizes is shown in Table 1.

```
void matrixInit()
{
    for(int row = 0 ; row < MatrixOrder ; row++ ) {
        for(int col = 0 ; col < MatrixOrder ;col++){
            srand(row+col);
            firstParaMatrix [row] [col] = ( rand() % 10 ) * FactorIntToDouble;
            secondParaMatrix [row] [col] = ( rand() % 10 ) * FactorIntToDouble;
        }
    }
}
```

Figure 11: Generating matrix code

Table 1: Time duration in different size of matrices

	128*128	256*256	512*512	1024*1024	2048*2048
Parallel	2ms	31ms	164ms	3491ms	43203ms
Serial	16ms	100ms	516ms	15584ms	134818ms

It will be more intuitive if we plot the results as a linear graph, as shown in figure 12. As we can see from the figure, the advantage of using OpenMP to multiply matrices increases as the matrix size increases. When the matrix size is small (order less than 500), the parallel algorithm has little gap with the serial algorithm. When the order reaches 1000 and 2000, the gap is very obvious.

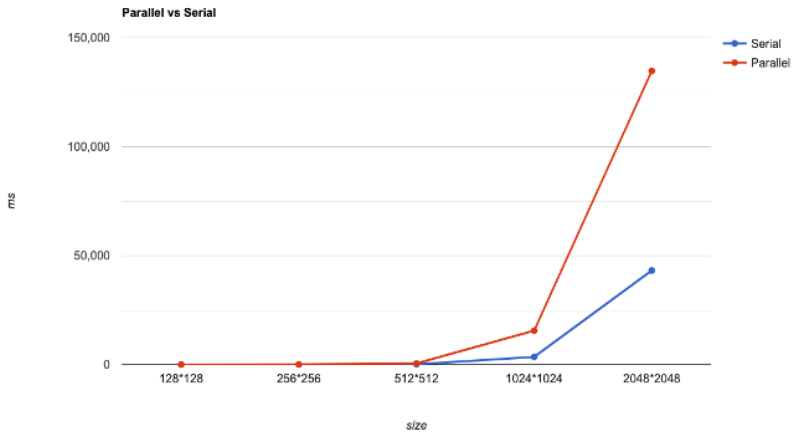

Figure 12: Comparison of time differences using the OpenMP and without OpenMP

As can be seen from the figure, as the matrix size increases, the advantage of parallel applications using OpenMP becomes more obvious. Therefore, it's necessary to use OpenMP for performance improvement. In the next section, we aim to utilize block matrix to minimize time complexity.

5.5 Use blocked matrix to optimize matrix multiplication

Blocked matrix multiplication is a technique in which you separate a matrix into different 'blocks' in which you calculate each block one at a time. This can be useful for larger matrices where spacial caching may come into play.[15]Below is an example of 3*3 block matrix multiplication.

$$\left[\begin{array}{cc|c} 1 & 2 & 3 \\ 4 & 5 & 6 \\ \hline 7 & 8 & 9 \end{array}\right] \left[\begin{array}{c|c} 1 & 2 \\ 3 & 4 \\ \hline 5 & 6 \end{array}\right] =$$

$$\begin{bmatrix} \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 1 \\ 3 \end{bmatrix} + \begin{bmatrix} 3 \\ 6 \end{bmatrix} [5] & \begin{bmatrix} 1 & 2 \\ 4 & 5 \end{bmatrix} \begin{bmatrix} 2 \\ 4 \end{bmatrix} + \begin{bmatrix} 3 \\ 6 \end{bmatrix} [6] \\ [7 \quad 8] \begin{bmatrix} 1 \\ 3 \end{bmatrix} + [9][5] & [7 \quad 8] \begin{bmatrix} 2 \\ 4 \end{bmatrix} + [9][6] \end{bmatrix}$$

The calculation of matrix multiplication is converted into the multiplication of its respective block matrices, which can effectively reduce the times of the multiplier matrix and the multiplier matrix into memory, and speed up the execution of the program. We tested the calculation speed of 2048*2048 matrix divided into four 512*512 matrices in a 16 threads environment.

```
void matrixMulti(int upperOfRow , int bottomOfRow ,
                 int leftOfCol , int rightOfCol ,
                 int transLeft ,int transRight )
{
    if ( ( bottomOfRow - upperOfRow ) < 512 )
        smallMatrixMult ( upperOfRow , bottomOfRow ,
                          leftOfCol , rightOfCol ,
                          transLeft , transRight );·

    else
    {
        #pragma omp task
        {
            matrixMulti( upperOfRow , ( upperOfRow + bottomOfRow ) / 2 ,
                         leftOfCol , ( leftOfCol + rightOfCol ) / 2 ,
                         transLeft , ( transLeft + transRight ) / 2 );
            matrixMulti( upperOfRow , ( upperOfRow + bottomOfRow ) / 2 ,
                         leftOfCol , ( leftOfCol + rightOfCol ) / 2 ,
                         ( transLeft + transRight ) / 2 + 1 , transRight );
        }

        #pragma omp task
        {
            matrixMulti( upperOfRow , ( upperOfRow + bottomOfRow ) / 2 ,
                         ( leftOfCol + rightOfCol ) / 2 + 1 , rightOfCol ,
                         transLeft , ( transLeft + transRight ) / 2 );
            matrixMulti( upperOfRow , ( upperOfRow + bottomOfRow ) / 2 ,
                         ( leftOfCol + rightOfCol ) / 2 + 1 , rightOfCol ,
                         ( transLeft + transRight ) / 2 + 1 , transRight );
        }
```

Figure 13: Block matrix multiplication

Figure 13 is the code of how we block the matrix. As you can see from the figure, if the size of the matrix is larger than 512*512, it does not calculate, but divides it into four smaller matrices of the same size and then repeats the cycle. We ran it five times and got the calculation time shown in Figure 14.

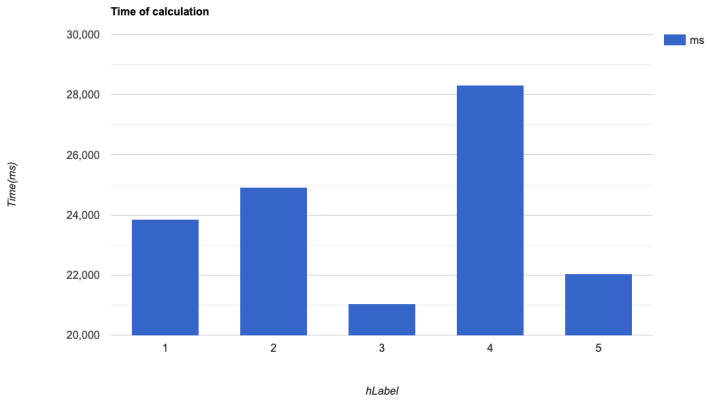

Figure 14: Block matrix multiplication

As can be seen from the figure, the time duration is around 20s and 30s, which significantly improves the speed compared with the matrix muplication without block operation which takes 43s. By blocking, we can take full advantage of parallel computing by breaking the task into smaller parts and .

6 MPI - an alternative API for parallel programming

6.1 Introduction

Message Passing Interface (MPI) is a standardized and portable message-passing standard designed to function on parallel computing architectures. OpenMPI is an implementation of the Message Passing Interface (MPI), used in distributed memory architectures.[5] The diagram below shows the difference of their architecture:

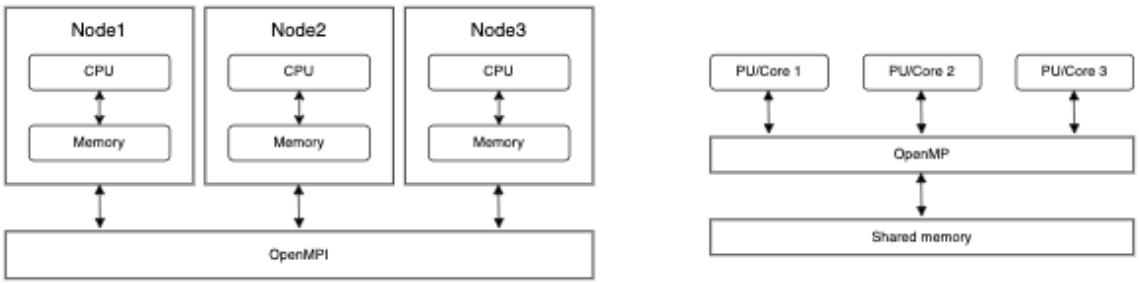

Figure 15: Block matrix multiplication

In this chapter, we explore the performance differences by replacing the Case Study 1 application that uses OpenMP to compute PI with one that uses MPI.

6.2 Experimental environment

The environment configuration we selected is as follows:
We rented two instances from AWS EC2 to build the MPI cluster, and how to set up a cluster environment can be found in my repository.

Table 2: Experimental environment

IP address	vCPU	Maximum thread	Operating System	AWS EC2 instances
18.188.176.222	4	4	Ubuntu 18.04 LTS	t2.2xlarge
18.122.126.211	4	4	Ubuntu 18.04 LTS	t2.2xlarge

6.3 Implementation with MPI

First, we need to launch an Amazon EC2 MPI cluster, the process can be found in my repository. Later, we changed the monte carlo program that used OpenMP to use MPI program. We set N=1000000000 and calculated the running time of the MPI program in single, dual, 4 and 8 threads, and compared the results with OpenMP programs under the same threads.

6.4 Experimental Evaluation

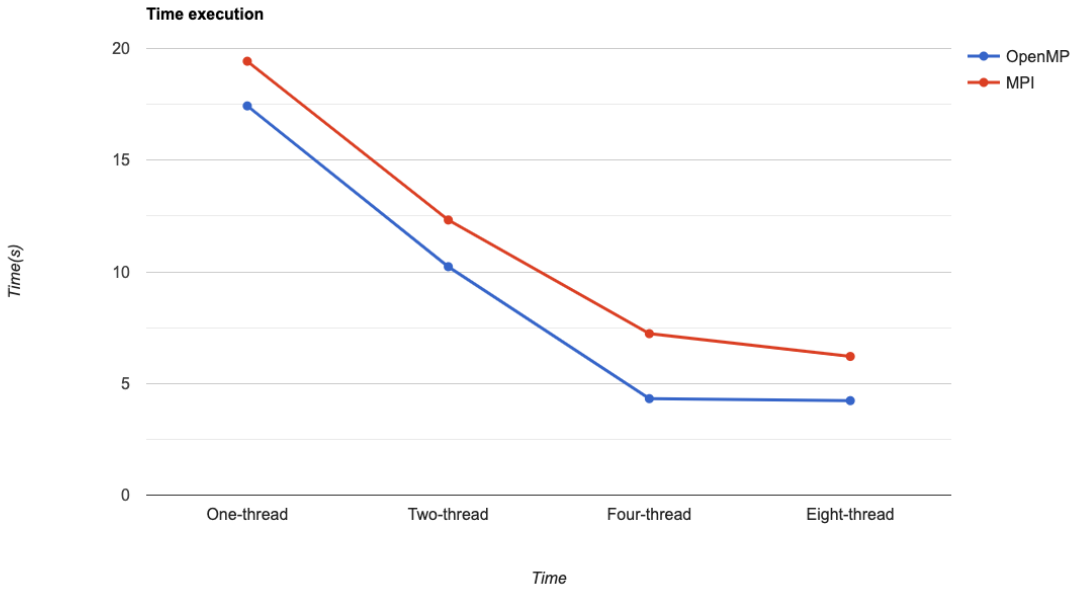

Figure 16: Block matrix multiplication

As shown in Figure 16, we can draw the following conclusions:

1. When running the same program, OpenMP is slightly faster than using MPI cluster computing. This is probably because MPI needs to send MPI messages for communicating between processes.

2. OpenMP only supports execution on shared memory, and we must consider using MPI when we want to run applications in distributed memory, even if there is a loss in message propagation.

3. In addition, we also found that programs using OpenMP don't need to change much of the original code, by simply adding some lines and could make application parallel, whereas MPI programs overhaul the original code.

7 Summary

In this paper, we conducted detailed survey on parallel programming and conclude why OpenMP becomes the standard API for shared-memory programming. Then we conducted two case studies for applications using OpenMP. In our result, we found by simply adding more threads will not always lead to performance improvement. In addition, by comparison with MPI, we summarized when we should use OpenMP for parallel programming and the drawbacks of OpenMP. We find that one thread with sufficient number of cores, OpenMP should be considered firstly for parallel programming, because it's slightly faster and no need to overhaul the original codes.

In the future studies, we want to answer the following questions: [1] Does different operating system will have any impact on parallel computing using OpenMP? [2] Is the performance the same when compiling OpenMP applications with different compilers? We have proved the effectiveness of OpenMP through two case studies in this paper and will explore the best practices of OpenMP in future studies.

Acknowledgement

Firstly, I would like to thank my professor, Prof. F.Dehne. He was my mentor. In his class, he taught us Parallel Computing and its applications. To be honest, I have no experience in parallel programming and even can't write C++ applications, so I really learned a lot from this course. I highly recommend [COMP 5704: Parallel Algorithms and Applications in Data Science] to those students who are interested in Parallel Programming.

During these three months, I went from a student who had no idea about parallel computing to a developer who could make parallel computing programs using OpenMP and MPI. During this period, professor also gave me some valuable advice. This is my last semester as a graduate student. I am very grateful that I chose this course to learn useful knowledge about Parallel Algorithms and Applications. I believe it will be helpful for my future career as a software engineer.

If you have any questions about my project, please contact me without hesitation, I would like to discuss it and hear your advices.

Many thanks to Prof. F.Dehne.

References

[1] K Mani Chandy. Parallel program design. In *Opportunities and Constraints of Parallel Computing*, pages 21–24. Springer, 1989.

[2] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*. MIT press, 2007.

[3] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[4] Stanley Gill. Parallel programming. *The computer journal*, 1(1):2–10, 1958.

[5] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[6] Robert L Harrison. Introduction to monte carlo simulation. In *AIP conference proceedings*, volume 1204, pages 17–21. American Institute of Physics, 2010.

[7] Frederick James. Monte carlo theory and practice. *Reports on progress in Physics*, 43(9):1145, 1980.

[8] Dirk P Kroese, Tim Brereton, Thomas Taimre, and Zdravko I Botev. Why the monte carlo method is so important today. *Wiley Interdisciplinary Reviews: Computational Statistics*, 6(6):386–392, 2014.

[9] MR Pimple and SR Sathe. Computing on multi-core platform: performance issues. In *Proceedings of the 2011 International Conference on Communication, Computing & Security*, pages 269–272, 2011.

[10] Michael J Quinn. Parallel programming. *TMH CSE*, 526:105, 2003.

[11] Thomas Rauber and Gudula Rünger. *Parallel programming*. Springer, 2013.

[12] Greg Slabaugh, Richard Boyes, and Xiaoyun Yang. Multicore image processing with openmp [applications corner]. *IEEE Signal Processing Magazine*, 27(2):134–138, 2010.

[13] Timothy Williamson. Calculating pi using the monte carlo method. *The Physics Teacher*, 51(8):468–469, 2013.

[14] Chao-Tung Yang, Chih-Lin Huang, and Cheng-Fang Lin. Hybrid cuda, openmp, and mpi parallel programming on multicore gpu clusters. *Computer Physics Communications*, 182(1):266–269, 2011.

[15] Fuzhen Zhang. Block matrix techniques. In *The Schur complement and its applications*, pages 83–110. Springer, 2005.

[16] Shanghong Zhang, Zhongxi Xia, Rui Yuan, and Xiaoming Jiang. Parallel computation of a dam-break flow model using openmp on a multi-core computer. *Journal of Hydrology*, 512:126–133, 2014.